

FIG. 1A

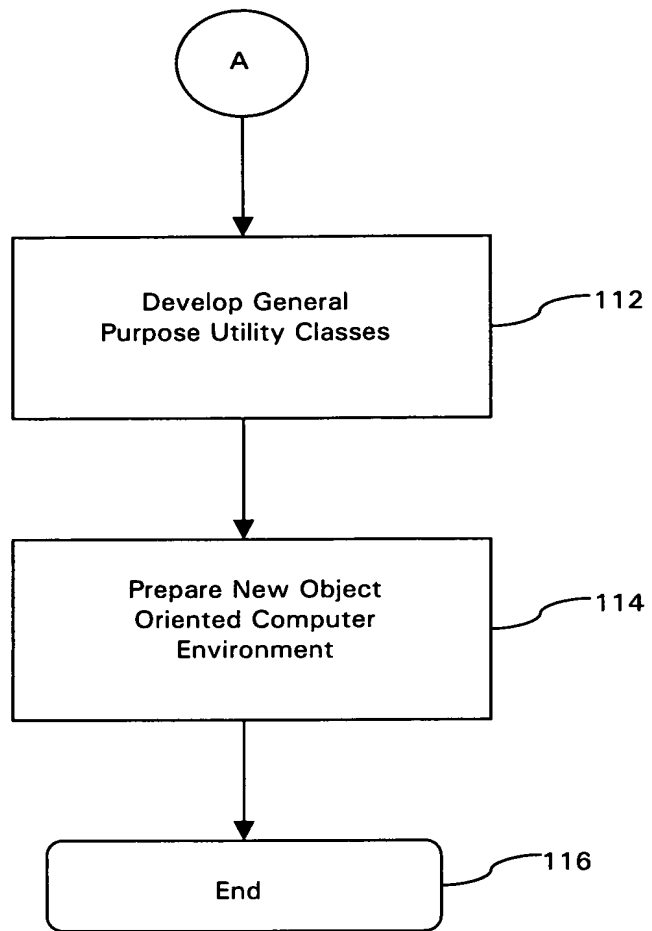


FIG. 1B

FIG. 2-1

```
// $Header: /2.0/Model/system.adn 27 5/15/98 3:20p Dan $
// System.adn - 05/15/98 09:45

// =====
// Model global controls (intended mainly for regression testing)
//   set to 2000 for release 2.0 (the default)
//   set to 1300 for regression testing against release 1.3
// =====
Constant DiskAssignmentAlgorithm      = 2000;
Constant RandomSeedAssignmentAlgorithm = 2000;
// =====
// Operating system interface constants (must match Strategizer internals)
// *** WARNING: changes in this section will cause execution time failure
// =====

// The Strategizer Operating System Model (a new feature for release 2.0)
// -----

// INTRODUCTION

// The operating system exists as a layer of software logic (and associated processes)
// that lies between software processes running in problem state (as in release 1.3)
// and the underlying hardware.

// An association is made between an operating system name (the first column in the
// CSE.ops file) and an ADN OS behavior name (the sixth column of the CSE.ops file).
// Note that the operating system names are selected from a list (based on the
// CSE.ops file) via the GUI for each computer in a Strategizer model.

// Instances of an operating system are created for each computer that runs an
// operating system with an associated ADN OS behavior name.
// The default for release 2.0 is to define the ADN behavior "ADNosSvc" for all named
// operating systems except "Generic" and "generic_operating_system".
// The operating system name is passed as a second parameter to the ADNosSvc behavior.

// A knowledgeable Strategizer user can create new operating system behaviors
// based on an understanding of the ADNosSvc behavior in this file (System.adn).
// Such user extensions must be added to the end of the System.adn file or included
// via an ADN Include statement at the end of the System.adn file.
```

FIG. 2-2

```

// A few words about the System.adn file. This file is loaded automatically at model
// initialization prior to the processing of ADN source generated or included by the GUI.
// A search is made of the directory containing the model first, then the installation area for
// the System.adn file. The location of the System.adn file selected is logged in the trace file.
// When modifications are planned, it is recommended that a copy of the System.adn file
// be made from the installation area to the directory containing the model.

// ADN PROCESSES AND STATE

// Software processes can execute in problem state and additionally in supervisor
// state (a new feature in release 2.0).

// Processes that startup in problem state switch to supervisor state at specific points
// (OS hook locations) to execute an operating system service and then return to problem state.

// Processes that startup in supervisor state (the OS server processes) remain in
// supervisor state.

// -----
// OS behavior hooks are implemented as cases of an ADN switch statement.
// The logic located at each hook is described along side the corresponding hook constant.
// -----

Constant INITIALIZEsvc = 0; // The INITIALIZEsvc hook is executed once for each associated
    {                       // computer by a special initialization process.
202                         // The purpose of this logic is to establish an operating system instance
                           // including its server processes and state data.
                           // Operating system data is maintained uniquely for each OS instance by using
                           // the functions osSetData and osGetData.
                           // Any user options and associated processing are included in this section.
                           // Refer to Case(INITIALIZEsvc) in the ADNosSvc behavior and the
                           // associated
                           // server behaviors ADNosNFS, ADNosVolumeMgr, and ADNosTaskMgr
                           // for additional
                           // information.

// In the following hooks the active process switches from problem to supervisor state and executes the top
// level (or main operating system behavior) in a manner very similar to a behavior call. The hook identifier

```

```
Constant TASKSTARTsvc =6; // The TASKSTARTsvc hook is executed when a Startup or Thread statement
// creates a new task (i.e, process or thread).
// The operating system task count is incremented. If the maximum number
// of tasks has already been reached, the creation of a new task is
// inhibited by blocking the current process (the requestor) until the task
// count drops below the maximum.
// Refer to Case(TASKSTARTsvc) in the ADNosSvc behavior and the ADNosTaskMgr
// behavior for additional details.
```

FIG. 2-4

```

Constant TASKENDsvc = 7; // The TASKENDsvc hook is executed whenever a process or thread terminates.
                        // The operating system task count is decremented. If the task count is
                        // greater than the maximum, the first blocked task is allowed to continue.
                        // Refer to Case(TASKENDsvc) in the ADNosSvc behavior and the ADNosTaskMgr
                        // behavior for additional details.

//-----
// The "hr" (hardware request data) utility functions are used to access specific data necessary
// to support the current operating system functionality. In release 2.0 this is limited to requests
// generated by the Execute statement.
// The constants defined below are used in combination with the following data access utility
// functions hrGetData/hrSetData to access scalar values, and hrGetDataX/hrSetDataX to access
// vector (or subscripted) values.

// CAUTION: In considering user defined extensions to the operating system the prospective user
// should become familiar with the data currently accessible at the ADN level.
// -----

Constant HExecSize = 4; //Used with hrGetData to obtain the size of the Execute request vector.
                        // The vector consists of the CPU request as first element (when present)
                        // followed by Read and/or Write requests elements.

Constant HExecReqType = 5; // Used with hrGetDataX to obtain the type of the Execute request element
                        // Returns one of the following: ReadType, WriteType, SendType, or CpuType.

Constant HRnfsProc = 6; // Used with hrSetData and hrGetData to save and retrieve the process id
                        // of the local NFS process.

Constant HRioReq = 7; // Used with hrGetDataX to obtain the handle to an IO request structure
                        // (element of the Execute vector).

Constant HResetReq = 8; // Used with hrSetDataX to set the specified element in the Execute request
                        // vector to null. This action is done when the original request element
                        // has been replaced by a more detailed operating system representation.

```

FIG. 2-5

```

Constant HRlocalIO = 9; // Used with hrSetData to initiate a local IO request using the specified
                        // IO request handle.

Constant HRpostExecute = 10; // Used with hrSetData to post a completion event to the original Execute
                             // synchronization control logic. (All parts of an Execute statement must
                             // be completed before a process exits the Execute statement.)

Constant HRkbytes = 11; // Used with hrGetDataX to obtain the total data bytes (in Kbytes) for the
                        // specified IO request

Constant HRvolumeHandle = 12; // Used with hrGetDataX to obtain the handle of the associated volume for
                              // LocVolType and RemVolType IO requests.
                              // The handle is used by volGetStripeSize() and volGetStripedDiskNumber()
                              // volume manager utility functions.

Constant HRkbytesOffset = 13; // Used with hrGetDataX to obtain the kbyte offset of the first IO record
                              // in the specified File based on the value of the FirstIO parameter on an
                              // execute Read or Write request. IfFirstIO is not specified a random
                              // record number between 0 and max-1 is used as the first IO record.
                              // The offset is used by the volume manager to determine the disk on which
                              // the first IO record resides.

Constant HRreqType = 14; // Used with hrGetDataX for Read and Write execute elements.
                        // Returns one of LocDiskType, RemDiskType, LocVolType, RemVolType.

Constant HRioReqDisk = 15; // Used with hrGetDataX to make a copy of the specified IO request

Constant HRioReqCopy = 16; // Used with hrSetDataX to set the disk number for the specified IO request

Constant HRioReqKbytes = 17; // Used with hrSetDataX to set the amount of data to be transferred

Constant HRioReqNumber = 18; // Used with hrSetDataX to set the starting record for the specified IO request

// -----
// Hardware request element type.
// Returned by HRexecReqType when used with hrGetDataX.

```

FIG. 2-6

```

// -----
Constant ReadType = 0;
Constant WriteType = 1;
Constant SendType = 2; // currently not needed
Constant CpuType = 3;

// -----
//IO request context type.
// Returned by HRreqType when used with hrGetDataX.
// -----
Constant LocDiskType = 0;
Constant RemDiskType = 1;
Constant LocVolType = 2;
Constant Rem VolType = 3;
Constant NonIoType = 4;

//=====
// ----- end of operating system interface constants -----
//=====

//miscellaneous parameters (used by ADNosNFS behavior)
// -----

Constant RPCreadReq = 40.0 / 1024.0; // kbytes
Constant RPCwriteAck = 40.0 / 1024.0; // kbytes

// task manager trace control (use for debugging only)
// -----
Constant TASKtrace = 0;

//=====
// default operating system service "main" behavior (referenced in CSE.ops)
// =====

Behavior ADNosSvc( svc_type, operating_system_name, computer_name, memory_structure,
    page_size, instr_per_page)
//NOTE: Only the "svc_type" parameter is available on all but the INITIALIZEsvc case.
Switch( svc_type ) {

```


FIG. 2-7

```

Case( INITIALIZEsvc ) {

//=====
// This logic is executed in 0 simulated time to initialize an instance of this
// operating system on each computer that specifies ADNosSvc in the CSE.ops file.
// The "operating_system_name" (second behavior parameter) corresponds to the

// name in column one of the CSE.ops file. This name may be used to differentiate
// between the initialization of differently named operating systems.
//=====

osSetData("svcState", 1); // required for initialization process

//-----
// Startup Memory Pageout Manager (required by memory model)
//-----
Startup proc = MemoryPageoutManager( memory_structure, page_size, instr_per_page)
    Priority 101;
processSetName(proc, "mpm-" + computer_name);

// OS service Master controls affect all operating system instances that
// specify use of the ADNosSvc behavior in column 6 of the CSE.ops file.

// active tasks control process
If ( osGetData("taskCountMax") >= 0 ) {
    osSetData("taskCount", 0);
    Startup proc = ADNosTaskMgr() Priority 101;
    osSetData("osTaskMgr", proc);
}

// NFS server process
Startup proc = ADNosNFS() Priority 101 Options "SetStatsFlag";
osSetData("osNFS", proc);
processSetName(proc, "nfs-" + computerGetName());

// volume manager
Startup proc = ADNosVolumeMgr() Priority 101;

osSetData("os VolMgr", proc);

```

FIG. 2-8

```

}

Case( EXECUTEsvc ) {

// this logic is executed in 0 simulated time to send any volume or remote IO requests
// included in an Execute statement to the local Volume manager or NFS server

execSize = hrGetData(HRexecSize);
i = 0;
While( i < execSize ) {
    reqType = hrGetDataX(HRexecReqType,i);
    Switch ( hrGetDataX(HRreqType,i) ) {
        Case( LocDiskType ) {
            // no OS service required
        }
        Case( RemDiskType ) {
            ioReq = hrGetDataX(HRioReq,i);
            Kbytes = hrGetDataX(HRkbytes,i);
            Send osGetData("osNFS") ("client_side",hrGetDataX(HRnfsProc,i),
                ioReq,reqType,Kbytes,0,0); // async
            hrSetDataX(HRresetReq,i,0);
        }
        Case( LocVolType ) {
            ioReq = hrGetDataX(HRioReq,i);
            Kbytes = hrGetDataX(HRkbytes,i);
            KbytesOffset = hrGetDataX(HRkbytesOffset,i);
            volumeHandle = hrGetDataX(HRvolumeHandle,i);
            Send osGetData(" os VolMgr") (0,ioReq,reqType,Kbytes,KbytesOffset,
                volumeHandle); // async
            hrSetDataX(HRresetReq,i,0);
        }
        Case( Rem VolType ) {
            ioReq = hrGetDataX(HRioReq,i);
            Kbytes = hrGetDataX(HRkbytes,i);
            kbytesOffset = hrGetDataX(HRkbytesOffset,i);
            volumeHandle = hrGetDataX(HRvolumeHandle,i);
            Send osGetData("osNFS") (" clientT_side",hrGetDataX(HRnfsProc,i),
                ioReq,reqType,Kbytes,

```

FIG. 2-9

```

        kbytesOffset,volumeHandle); // async
        hrSetDataX(HRresetReq ,i,0);

    }
    Case( NonIoType ) {
        // no OS service required

    }
    }
    i = i + 1;
}
}
Case( SENDsvc ) {
    // Execute Cpu 0.000001;
}
Case( SENDWAITsvc ) {
    // Execute Cpu 0.000001;
}
Case( RECEIVESvc ) {
    // Execute Cpu 0.000001;
}
Case( REPLYsvc ) {
    // Execute Cpu 0.000001;
}
Case( TASKSTARTsvc ) {

    // increment task count
    taskCount = osGetData("taskCount") + 1;
    osSetData("taskCount",taskCount);

    // if task count exceeds max put new task in task manager's queue
    // and put new task into wait state
    If (taskCount > osGetData("taskCountMax") ) {
        Send osGetData(" osT askMgr") (threadGetCurrentId());
        If ( TASKtrace ) {
            Print stringFormat("%.6f",simGetTime()),
                "ADNosTaskMgr: task",threadGetCurrentId()," suspended";
        }
        threadWaitForSignal();
    }
}
}

```

FIG. 2-10

```

    }
    Case( TASKENDsvc ) {

        // decrement task count
        taskCount = osGetData("taskCount") - 1;
        osSetData("taskCount",taskCount);

        // if there is a waiting task, signal task manager
        If ( taskCount >= osGetData("taskCountMax") ) {
            processSignal( osGetData("osTaskMgr") );
        }
    }
    Return( svc_type );
}

// Maximum number of active tasks manager behavior
//-----

Behavior ADNosTaskMgr( ) {
    While( 1 ) {
        // wait for signal from TASKENDsvc
        process WaitForSignal();

        // remove first task from input queue and signal it
        Receive( task_id ) {
            If ( TASKtrace ) {
                Print stringFormat(" %.6f",simGetTime()),
                    "ADNosTaskMgr: task", task_id, "resumed";
            }
            threadSignal( task_id );
        } Reply();
    }
}

// NFS server behavior
// -----
Behavior ADNosNFS( ) {
    osSetData(" svcState", 1);
    processSetNoThreadUtilizationStats();
}

```

FIG. 2-11

```

While( 1 ) {
    Receive(type,arg1,arg2,arg3,arg4,arg5,arg6) Thread {
        Switch( type) {
            Case( "client_side" ) {
                // save client process id
                execute_proc = messageGetSendingProcessId();
                processSetClientProcessId( execute_proc ); // c_proc->client_proc_sn =
                                                            execute_proc

                If ( arg3 == ReadType ) {
                    msgSendLength = RPCreadReq;
                    msgReplyLength = arg4;
                }
                Else { // WriteType
                    msgSendLength = arg4;
                    msgReplyLength = RPCwriteAck;
                }

                // forward request to remote server
                // ( msg->client_proc_sn = c_proc->client_proc_sn)
                Send arg1 ( " server_side" ,arg2,arg3,arg4,arg5,arg6,execute_proc) Message
                                                            msgSendLength

                Protocol "UDP/IP" Wait();

                // post completion event to Execute statement synchronization control
                If ( ! arg6 ) { // not a volume manager request
                    hrSetData(HRpostExecute,execute_proc) ;
                }
            }
            Case( "server_side" ) {
                processSetClientProcessId( arg6 ); // execute_proc
                If ( arg5 ) {
                    // volume request
                    Send osGetData("osVolMgr") (arg6,arg1,arg2,arg3,arg4,arg5); // async
                }
                Else {

                    // disk request
                    If ( arg2 == ReadType ) {
                        msgReplyLength = arg3;
                    }
                    Else { // WriteType
                        msgReplyLength = RPCwriteAck;
                    }

                    // issue local IO request hrSetData(HRlocalIO,arg 1);
                }
            }
        }
    } Reply() Message msgReplyLength;
}

```

FIG. 2-12

```

Behavior ADNoVolumeMgr() }
osSetData("svcState", 1);
processSetNoThreadUtilizationStats()
While(1) {
    Receive( execute_proc, io_req, req_type, req_kbytes, first_kbytes_offset, volume_handle) Thread {

        If ( !execute_proc ) { // local request
            execute_proc = messageGetSendingProcessId();
        }
        processSetClientProcessId( execute_proc );
        // collect statistics
        volBeginRequest( volume_handle, execute_proc );
        request_start_time = simGetTime();

        // for each volume IO request in Execute statement
        kbytes_offset = first_kbytes_offset;
        kbytes = req_kbytes; // total bytes in this I/O request ( Bytes * Number)
        stripe_kbytes = volGetStripeSize(volume_handle);

        // process first stripe, partial stripe up to a stripe boundary, or full request
        mod_kbytes_offset = RMod(kbytes_offset,stripe_kbytes);
        curr_kbytes = RMin(stripe_kbytes-mod_kbytes_offset,kbytes);

        // modify Number field of original request (to avoid setting it each time)
        hrSetDataX(HRioReqNumber,io_req, 1);

        // save the disk number as the reference point for a complete pass through
        // all of the disks in the volume
        first_disk_number = vol GetStripedDiskNumber( volume_handle,kbytes_offset);

```

Join

FIG. 2-13

```

// loop until all of the data has been processed
While ( kbytes > 0.0005 ) {
    Join {

        disk_number = volGetStripedDiskNumber( volume_handle,kbytes_offset);

        // loop over each disk on volume once while there is more data

        While ( ( disk_number >= 0 ) && ( kbytes > 0.0005 ) ) {

            // for each piece of an I/O request
            Thread {
                // declare client process for associating statistics
                processSetClientProcessId( execute_proc );

```

FIG. 2-14

```

If ( kbytes > curr_kbytes ) {
// copy original I/O request
ioReq = hrGetDataX(HRioReqCopy,io_req);
}
Else {
// use original I/O request
ioReq = io_req;
}

// modify selected fields
hrSetDataX(HRioReqDisk,ioReq,disk_number );
hrSetDataX(HRioReqKbytes,ioReq,curr_kbytes );

// issue local IO request
hrSetData(HRlocalIO,ioReq);
}
kbytes_offset = kbytes_offset + curr_kbytes;
kbytes = kbytes - curr_kbytes;
curr_kbytes = RMin(stripe_kbytes,kbytes);

disk_number = volGetStripedDiskNumber( volume _handle,kbytes- offset);

If ( disk_number = first_disk_number ) {
    disk_number = -1;
}
} // While - loop over each disk on the volume once while there is more data
} // Join - wait here until all the disks have completed
} // While - loop while there is more data to be processed
} // Join - wait here until all the data has been processed and all of the threads completed

// post completion event to Execute statement synchronization control
// when all pieces of this request have been completed
hrSetData(HRpostExecute,execute_proc) ;

// collect statistics
volEndRequest( volume_handle, execute_proc, request_start_time);
} Reply();
}
}

// user defined OS behavior include statements
//-----

// Include "user_OS_behaviors.adn"; // <= = sample syntax

```

FIG. 3-1

```

// $Header: /ST/Trunk/Model/system.adn 81 12/19/00 1:11p Dan $
// System.adn - 12/21/2000 09:00

// -----
// Copyright Hyperformix, Inc., 1996-2000.
// This software, including the program, help files and documentation, is
// owned by Hyperformix, Inc.
// The software contains information which is confidential and proprietary
// to Hyperformix, Inc. Access to and use of the software is available only
// through a nonexclusive license agreement with Hyperformix, Inc.
// The use of this software is controlled by that license agreement and any
// other use or copying of the software will violate the license and is
// expressly prohibited.
// -----

package "OperatingSystemPackage"; ~ 312

//=====

// Operating system interface constants (must match Strategizer internals)
// *** WARNING: changes in this section will cause execution time failure ~ 320
//=====

// The Strategizer Operating System Model has been modified for release 2.2
// to take advantage of the new ADN object-oriented extensions.
// Strategizer users can extend this SES supplied capability by using the
// the new user_extensions.adn option.
// -----

// INTRODUCTION

// The operating system exists as an instance of the class ses_OperatingSystem
// and a layer of software logic implemented in its behavior methods and
// associated server processes. This layer of logic lies between software
// processes running in application problem state and the underlying hardware.

// An association is made between an operating system name (the first column in the
// CSE.ops file) and an ADN OS behavior name (the sixth column of the CSE.ops file).
// Note that the operating system names are selected from a list (based on the
// CSE.ops file) via the GUI for each computer in a Strategizer model.

```


FIG. 3-2

```

// Instances of an operating system are created for each computer that runs an
// operating system with an associated ADN OS behavior name by invoking that
// behavior to instantiate an OperatingSystem object and call its
// initializeSvc behavior.
// The default for this release is to define the ADN behavior "ADNosSvc" for
// all the named operating systems. The operating system name is passed as a
// parameter to the operating system instance constructor.

// A knowledgeable Strategizer user can create a new operating system class by
// extending the OperatingSystem class supplied by SES in this file (System.adn).
// Such user extensions must be placed in the specially named user_extensions.adn
// file for proper processing.

// ADN PROCESSES AND STATE

// Software processes can execute in problem state and additionally in supervisor
// state (a new feature since release 2.0).

// Processes that startup in problem state switch to supervisor state at specific points
// (OS hook locations) to execute an operating system service and then return to problem
// state.

// Processes that startup in supervisor state (the OS server processes) remain in
// supervisor state.

// -----
// OS behavior hooks are implemented as methods of an instance of the ses_OperatingSystem class
// or a user extension thereof specified in the user_extensions.adn file.
// The logic located at each hook is described along side the corresponding hook constant.
// Note: The hook constant is required on the return from each method as part of the hook
// protocol mechanism.
// -----

Constant INITIALIZEsvc = 0; // The initializeSvc behavior is executed once for each associated
    }                      // computer by a special initialization process after a new instance
302                        // of the ses_OperatingSystem class is created. These actions are taken
                          // by the ADN OS behavior (named in col. 6 of the CSE.ops file).
                          // The purpose of this logic is to create the associated server

```

FIG. 3-3

```

// processes that make up part of the operating system.
// The operating system state data is maintained in the OperatingSystem
// instance field variables.
// The initializeSvc behavior of the ses_OperatingSystem class should
// be called as the first statement in any initializeSvc overriding
// behavior specified by the user to assure that the basic operating
// system services are properly initialized.
// Refer to the initializeSvc behavior logic for additional details.

// In the following hooks the active process switches from problem to supervisor state and executes the
// corresponding operating system service behavior. The hook constant value is passed back as the only
// return parameter. When the service is completed, the active process returns to problem state.

Constant EXECUTEsvc = 1; // The executeSvc behavior receives control when the Execute statement is ready
                        // to be sent to the hardware.
                        // Individual elements in the request vector (prepared from the Execute
                        // statement) are checked for remote disk IO and IO operations involving
                        // files located on volumes. Substitution or modification of the original
                        // requests are made as appropriate. The requests are then passed on to
                        // the hardware model.
                        // Refer to the executeSvc behavior logic for additional details.
                        // Note: It is strongly recommended that this behavior not be overridden
                        // by the user unless all the original logic is also included.

// The following set of four hooks are designed to work together to provide support for the implementation
// of communication protocol logic. This is expected to be the main part of the operating system logic that
// most users may be interested in extending.
// The service behaviors provided with release 2.2 contain no logic other than to surface addressability
// to the ses_Message object instance associated with the operation. The declaration for the ses_Message class
// is located in the Utilites.adn file.

// The following notes may help in use of the communication service hooks:
// - Synchronous messages execute the following sequence: sendSvc, receiveSvc, replySvc, and sendWaitSvc.
// - Asynchronous messages execute the following sequence: sendSvc then receiveSvc.
// - The sendSvc and replySvc are invoked just before passing control to the hardware.
// - The receiveSvc and sendWaitSvc are invoked just after returning from the hardware.

Constant SENDSvc = 2; // The sendSvc behavior is executed at the end of the software part
                      // of Send statement processing, just before the request is sent to the
                      // hardware. Upon exit from this section, the resulting request is sent
                      // to the hardware.

```

FIG. 3-4

```

Constant SENDWAITsvc = 3; // The sendWaitSvc behavior is executed early in the processing of an incoming
                          // message sent by the Reply clause of a Receive statement.
                          // Upon exit from this section, control is passed to the Wait clause of the
                          // original Send statement for processing of the message data fields.

Constant RECEIVESvc = 4; // The receiveSvc behavior is executed early in the processing of an incoming
                          // message from a Send statement.
                          // Upon exit from this section, control is passed to the Receive statement
                          // for processing of the message data fields.

Constant REPLY svc = 5; // The replySvc behavior is executed at the end of the software part
                        // of the Reply clause (part of the Receive statement), just before the
                        // request is sent to the hardware.

Constant TASKSTARTsvc = 6; // Updates active task count stats
                           // Increments active task count
                           // Issues warning first time maximum count is issued

Constant TASKENDsvc = 7; // Updates active task count stats
                          // Decrements active task count

// -----
// The "hr" (hardware request data) utility functions are used to access specific data necessary
// to support the current operating system functionality. In release 2.0 this is limited to requests
// generated by the Execute statement.
// The constants defined below are used in combination with the following data access utility
// functions hrGetData/hrSetData to access scalar values, and hrGetDataX/hrSetDataX to access
// vector (or subscripted) values.

// CAUTION: In considering user defined extensions to the operating system the prospective user
// should become familiar with the data currently accessible at the ADN level.
// -----

Constant HExecSize = 4; // Used with hrGetData to obtain the size of the Execute request vector.
                        // The vector consists of the CPU request as first element (when present)
                        // followed by Read and/or Write requests elements.

Constant HExecReqType = 5; // Used with hrGetDataX to obtain the type of the Execute request element.
                           // Returns one of the following: ReadType, WriteType, SendType, or CpuType.

```

FIG. 3-5

Constant HRnfsProc = 6;	// Used with hrSetData and hrGetData to save and retrieve the process id // of the local NFS process.
Constant HRioReq = 7;	// Used with hrGetDataX to obtain the handle to an IO request structure // (element of the Execute vector).
Constant HRresetReq = 8;	// Used with hrSetDataX to set the specified element in the Execute request // vector to null. This action is done when the original request element // has been replaced by a more detailed operating system representation.
Constant HRlocalIO = 9;	// Used with hrSetData to initiate a local IO request using the specified // IO request handle.
Constant HRpostExecute = 10;	// Used with hrSetData to post a completion event to the original Execute // synchronization control logic. (All parts of an Execute statement must // be completed before a process exits the Execute statement.)
Constant HRkbytes = 11;	// Used with hrGetDataX to obtain the total data bytes (in Kbytes) for the // specified IO request
Constant HRvolumeHandle = 12;	// Used with hrGetDataX to obtain the handle of the associated volume for // LocVolType and RemVolType io requests. // The handle is used by volGetStripeSize() and volGetStripedDiskNumber() // volume manager utility functions.
Constant HRkbytesOffset = 13;	// Used with hrGetDataX to obtain the kbyte offset of the first IO record // in the specified File based on the value of the FirstIo parameter on an // execute Read or Write request. If FirstIo is not specified a random // record number between 0 and max-1 is used as the first IO record. // The offset is used by the volume manager to determine the disk on which // the first IO record resides.
Constant HRreqType = 14;	// Used with hrGetDataX for Read and Write execute elements. // Returns one of LocDiskType, RemDiskType, LocVolType, RemVolType.
Constant HRioReqCopy = 15;	// Used with hrGetDataX to make a copy of the specified IO request
Constant HRioReqDisk = 16;	// Used with hrSetDataX to set the disk number for the specified IO request

FIG. 3-6

```

Constant HRioReqKbytes = 17; // Used with hrSetDataX to set the amount of data to be transferred

Constant HRioReqNumber = 18; // Used with hrSetDataX to set the starting record for the specified IO request

Constant HRlocalVIO = 19; // Used with hrSetData to initiate a local volume manager IO request using the specified
                          // IO request handle and applying physical attribute.

// -----
// Hardware request element type.
// Returned by HRexecReqType when used with hrGetDataX.
// -----
Constant ReadType = 0;
Constant WriteType = 1;
Constant SendType = 2; // currently not needed
Constant CpuType = 3;

// -----
// IO request context type.
// Returned by HRreqType when used with hrGetDataX.
// -----
Constant LocDiskType = 0;
Constant RemDiskType = 1;
Constant Loc VolType = 2;
Constant Rem VolType = 3;
Constant NonIoType = 4;

// =====
// end of operating system interface constants -----
// =====

// miscellaneous parameters (used by ADNosNFS behavior)
// -----

Constant RPCreadReq = 40.0 | 1024.0; // kbytes
Constant RPCwriteAck = 40.0 | 1024.0; // kbytes

```

FIG. 3-7

```

// task manager trace control (use for debugging only)
// -----
Constant TASKtrace = 0;

// -----
// remote IO distribution policy - used by NFS servers
// -----

public associative gRemoteloDistributionPolicy[100];

public function registerRemoteloDistributionPolicy( tComputerName,
userRemoteloDistributionPolicyName) {
    gRemoteloDistributionPolicy[ tComputerName] = userRemoteloDistributionPolicy Name;
}

// =====
// default operating system service "main" behavior (referenced in CSE.ops)
// =====

public class ses_OperatingSystem {
    static integer    fTaskMaxWarningIssued = false;
    static associative fActiveTaskCountStatsPtr[100];
    string            fOpSysName;
    string            fComputerName;
    integer            fMemoryStruct;
    real              fPageSize;
    real              fInstrPerPage;
    integer            fOsMemMgr;
    string            fRemoteloDistributionPolicy;
    integer            fOsNFS;
    integer            fOsTaskMgr;
    integer            fTaskCountMax = -1;
    integer            fTaskMaxReached = false;
    ses_Statistic      fActiveTaskCountStats = null;
    integer            fActiveTaskCount = 0;
    integer            fOsVolMgr;

```

304

310

314

316

306

FIG. 3-8

```

ses_ThreadList  fThreadList;

constructor ses_OperatingSystem(aOpSysName,aComputerName,aMemoryStruct,
                                aPageSize,aInstrPerPage) {
    fOpSysName = aOpSysName;
    fComputerName = aComputerName;
    fMemoryStruct = aMemoryStruct;
    fPageSize = aPageSize;
    fInstrPerPage = aInstrPerPage;
    fRemoteloDistributionPolicy = gRemoteloDistributionPolicy[stringNameBase(aComputerName)];
}

behavior initializeSvc() {

    //=====
    // This logic is executed in 0 simulated time to initialize an instance of this
    // operating system on each computer that specifies ADNosSvc in the CSE.ops file.
    // The "operating_system_name" (second behavior parameter) corresponds to the
    // name in column one of the CSE.ops file. This name may be used to differentiate
    // between the initialization of differently named operating systems.
    //=====

    osSetData("operatingSystemInstance",this);

    // osSetData("svcState",1);          // required for initialization process

    // -----
    // Startup Memory Pageout Manager (required by memory model)
    // (use priority of 100 for compatibility with rel 2.1)
    // -----
    Startup fOsMemMgr = MemoryPageoutManager( fMemoryStruct, fPageSize,
                                              fInstrPerPage ) Priority 100;
    processSetName(fOsMemMgr,"mpm-"+fComputerName);

    // OS service Master controls affect all operating system instances that
    // specify use of the ADNosSvc behavior in column 6 of the CSE.ops file.

    // active tasks control process

```

FIG. 3-9

```

fTaskCountMax = osGetData("taskCountMax");
If ( fTaskCountMax >= 0 ) {
    Call initTaskMgr();
}

// NFS server process
Startup fOsNFS = ADNosNFS( this ) Priority 100 Options "NoStatsFlag";
processSetNameOnly(fOsNFS, "nfs-"+computerGetName());
registerSendDistributionPolicy2( fOsNFS, fRemoteloDistributionPolicy );

// volume manager
Startup fOsVolMgr = ADNosVolumeMgr() Priority 100;
}

behavior executeSvc() {
    //=====
    // this logic is executed in 0 simulated time to send any volume or remote IO requests
    // included in an Execute statement to the local Volume manager or NFS server
    //=====
    variable i;           // index variable
    variable execSize;     // number of request elements in the execute statement
    variable reqType;      // request element type: LocDisk, RemDisk, LocVol, RemVol, Nonlo
    variable ioReq;        // I/O request handle
    variable Kbytes;       // Size in Kbytes of an I/O request
    variable KbytesOffset; // Offset in file of first byte of data
    variable volumeHandle; // Handle to volume where I/O data is located

    execSize = hrGetData(HRexecSize);
    i = 0;
    While( i < execSize ) {
        reqType = hrGetDataX(HRexecReqType,i);
        Switch ( hrGetDataX(HRreqType,i) ) {
            Case( LocDiskType ) {
                // no OS service required
            }
            Case( RemDiskType ) {
                ioReq = hrGetDataX(HRioReq,i);
                Kbytes = hrGetDataX(HRkbytes,i);
                Send fOsNFS ("client_side",hrGetDataX(HRnfsProc,i),
                    ioReq,reqType,Kbytes,0,0); // async
            }
        }
        i++;
    }
}

```

308

FIG. 3-10

```

        hrSetDataX(HRresetReq,i,0);
    }
    Case( LocVolType ) {
        ioReq = hrGetDataX(HRioReq,i);
        Kbytes = hrGetDataX(HRkbytes,i);
        KbytesOffset = hrGetDataX(HRkbytesOffset,i);
        volumeHandle = hrGetDataX(HRvolumeHandle,i);
        Send fOs VolMgr (0,ioReq,reqType,Kbytes,KbytesOffset,
            volumeHandle,0); // async
        hrSetDataX(HRresetReq,i,0);
    }
    Case( RemVolType ) {
        ioReq = hrGetDataX(HRioReq,i);
        Kbytes = hrGetDataX(HRkbytes,i);
        KbytesOffset = hrGetDataX(HRkbytesOffset,i);
        volumeHandle = hrGetDataX(HRvolumeHandle,i);
        Send fOsNFS ("client_side",hrGetDataX(HRnfsProc,i),
            ioReq,reqType,Kbytes,
            KbytesOffset,volumeHandle); // async
        hrSetDataX(HRresetReq,i,0);
    }
    Case( NonIoType ) {
        // no OS service required
    }
    }
    i = i + 1;
}
return( EXECUTEsvc );
}

//=====
// The logic in the following four behaviors: sendSvc, sendWaitSvc, receiveSvc, replySvc
// is invoked on all application state logic originating from send/wait receive/reply
// ADN statements
//=====
behavior sendSvc(aMsg) {
    variable tMsg;
    tMsg = ses_Message.associatedMsg( aMsg );
    // < Insert optional logic here>
    tMsg.sendToHardware(tMsg.receiving_proc_sn,tMsg.message_bytes);
}

```

FIG. 3-11

```

    return( SENDsvc );
}
behavior sendWaitSvc(aMsg) {
    variable tMsg;
    tMsg = ses_Message.associatedMsg( aMsg );
    // < Insert optional logic here >
    return( SENDWAITsvc );
}
behavior receiveSvc(aMsg) {
    variable tMsg;
    tMsg = ses_Message.associatedMsg( aMsg );
    // < Insert optional logic here >
    return( RECEIVESvc );
}
behavior replySvc(aMsg) {
    variable tMsg;
    tMsg = ses_Message.associatedMsg( aMsg );
    tMsg.sendToHardware(tMsg.receiving_proc_sn,tMsg.message_bytes);
    // < Insert optional logic here >
    return( REPLY svc );
}

//=====
// Maximum task control management
// o Keeps track of all active threads executing on computing node
// o Is controlled via the corresponding entry in the CSE.ops file
//=====

// ----- logic for release 3.0

behavior taskStartSvc( thid ) {
    fActiveTaskCount = fActiveTaskCount + 1;
    fActiveTaskCountStats.sample(1.0);
    If ( fTaskMaxReached == false ) {
        If (fActiveTaskCount == fTaskCountMax) {
            if ( fTaskMaxWarningIssued == false ) {
                fTaskMaxWarningIssued = true;
                Warning "***** First maximum concurrent task count reached.\n",
                " Check trace file for time of first occurrence and computer name for each computer.\n",
                " Check report file \"Custom Statistics\" for active task count statistics for each computer. ";
            }
        }
    }
}

```

FIG. 3-12

```

    }
    fTaskMaxReached = true;
    Print stringFormat("%.6f",simGetTime()),
    "***** Maximum concurrent task count limit reached for computer",
    "\"" + fComputerName + "\" ";
}
}
return( TASKSTARTsvc );
}

behavior taskEndSvc( thid ) {
    fActiveTaskCount = fActiveTaskCount - 1;
    fActiveTaskCountStats.sample(-1.0);
    return( TASKENDsvc );
}

behavior initTaskMgr() {
    // create active task count user stat
    tStatsName = ses_ComputerStatName(fComputerName);
    if ( associativeArrayElementIsDefined(fActiveTaskCountStatsPtr,tStatsName) ) {
        fActiveTaskCountStats = fActiveTaskCountStatsPtr[tStatsName];
    }
    else {
        fActiveTaskCountStats = ses_gStatMgr.createContinuousStatistic
        ("TaskMgr_activeTasks_" + tStatsName);
        fActiveTaskCountStatsPtr[tStatsName] = fActiveTaskCountStats;
    }
}

}

// -----
// NFS server behavior
// -----

Behavior ADNosNFS( aServer ) {
    variable tExecuteProc;
    real tMsgSendLength;
    real tMsgReplyLength;

    osSetData("svcState", 1);

```

FIG. 3-13

```

processSetNoThreadUtilizationStats();

While( 1 ) {
    Receive( aType,arg1,arg2,arg3,arg4,arg5,arg6 ) Thread {
        Switch( aType ) {
            Case( "client_side" ) {
                // save client process id
                tExecuteProc = messageGetSendingProcessId();
                processSetClientProcessId( tExecuteProc ); // c_proc->client_proc_sn = execute_proc
                If ( arg3 == ReadType ) {
                    tMsgSendLength = RPCreadReq;
                    tMsgReplyLength = arg4;
                }
                Else { // WriteType
                    tMsgSendLength = arg4;
                    tMsgReplyLength = RPCwriteAck;
                }

                // forward request to remote server
                send arg1("server_side",arg2,arg3,arg4,arg5,arg6,tExecuteProc) Message tMsgSendLength
                Protocol "UDP/IP" Wait();

                // post completion event to Execute statement synchronization control
                // If ( ! arg6 ) { // not a volume manager request -- bug 3225 fix

                hrSetData(HRpostExecute,tExecuteProc);

                // }
            }
            Case( "server_side" ) {
                processSetClientProcessId( arg6 ); // execute_proc
                If ( arg5 ) {
                    // volume request
                    // tMsgReplyLength = 0.0;
                    tMsgReplyLength = arg3; // bug 3225 fix
                    Send (aServer.fOs VolMgr) (arg6,arg1,arg2,arg3,arg4,volGetLocalHandle(arg5),
                    threadGetCurrentId()); //
                }
                Else {
                    threadWaitForSignal();
                }
            }
        }
    }
}

```

FIG. 3-14

```

// disk request
If ( arg2 == ReadType ) {
    tMsgReplyLength = arg3;
}
Else { // WriteType
    tMsgReplyLength = RPCwriteAck;
}
// issue local IO request
hrSetData(HRlocalIO,arg1);
}
}
}
} Reply() Message tMsgReplyLength;
}
}

//-----
// Volume manager behavior
// -----

Behavior ADNosVolumeMgr() {

    // thread variables (separate copy for each)
    variable kbytes;
    variable kbytes_offset;
    variable request_start_time;
    variable stripe_kbytes;
    variable mod_kbytes_offset;
    variable curr_kbytes;

    osSetData("svcState",1);
    processSetNoThreadUtilizationStats();

    While( 1 ) {
        Receive( execute_proc, io_req, req_type, req_kbytes, first_kbytes_offset,
            volume_handle, waitId ) Thread {

            osSetData("svcState",1);

            If ( !execute_proc ) { // local request
                execute_proc = messageGetSendingProcessId();
            }
        }
    }
}

```

FIG. 3-15

```

}
processSetClientProcessId( execute_proc );

// collect statistics
volBeginRequest( volume_handle, execute_proc );
request_start_time = simGetTime();

// for each volume IO request in Execute statement
kbytes_offset = first_kbytes_offset;
kbytes = req_kbytes; // total bytes in this I/O request ( Bytes * Number )
stripe_kbytes = volGetStripeSize(volume_handle);

// process first stripe, partial stripe up to a stripe boundary, or full request
mod_kbytes_offset = RMod(kbytes_offset,stripe_kbytes);
curr_kbytes = RMin(stripe_kbytes-mod_kbytes_offset,kbytes);

// modify Number field of original request (to avoid setting it each time)
hrSetDataX(HRioReqNumber,io_req,1);

// save the disk number as the reference point for a complete pass through
// all of the disks in the volume
first_disk_number = volGetStripedDiskNumber( volume_handle,kbytes_offset);

Join {

    // loop until all of the data has been processed

    While ( kbytes > 0.0005 ) {

        Join {

            disk_number = volGetStripedDiskNumber(volume_handle,kbytes_offset);

            // loop over each disk on volume once while there is more data

            While ( ( disk_number >= 0 ) && ( kbytes > 0.0005 ) ) {

                // for each piece of an I/O request
                Thread {
                    // declare client process for associating statistics

```

FIG. 3-16

```

processSetClientProcessId( execute_proc );

If ( kbytes > curr_kbytes ) {
    // copy original I/O request
    ioReq = hrGetDataX(HRioReqCopy,io_req);
}
Else {
    // use original I/O request
    ioReq = io_req;
}

// modify selected fields
hrSetDataX(HRioReqDisk,ioReq,disk_number) ;
hrSetDataX(HRioReqKbytes,ioReq,curr_kbytes);

// issue local IO request
hrSetData(HRlocalVIO,ioReq);
}
kbytes_offset = kbytes_offset + curr_kbytes;
kbytes = kbytes - curr_kbytes;
curr_kbytes = RMin(stripe_kbytes,kbytes);

disk_number = volGetStripedDiskNumber(volume_handle,kbytes_offset);
If ( disk_number == first_disk_number ) {
    disk_number = -1;
}
} // While - loop over each disk on the volume once while there is more data

} // Join - wait here until all the disks have completed

} // While - loop while there is more data to be processed

} // Join - wait here until all the data has been processed and all of the threads completed

// post completion event to Execute statement synchronization control
// when all pieces of this request have been completed

if ( waitId ) {
    // request from NFS
    threadSignal(waitId);
}

```

FIG. 3-17

```

    }
    else {
        // local request
        hrSetData(HRpostExecute,execute_proc) ;
    }

    // collect statistics
    volEndRequest( volume_handle, execute_proc, request_start_time );

    } Reply();
}

//=====
// -----
// this is a required operating system factory behavior
// its name should appear in column 6 of the CSE.ops file for all
// named operating systems that use the OperatingSystem class
// -----

public behavior ADNosSvc( aSvcType, aArg2, aComputerName,
                        aMemoryStructure, aPageSize, alnstrPerPage ) {
    variable t_OpSys;
    if ( aSvcType == 0 ) {
        tOpSys = new ses_OperatingSystem( aArg2, aComputerName,
            aMemoryStructure, aPageSize, alnstrPerPage );
        call tOpSys.initializeSvc();
    }
}

```

} 318